The Universal Design Pattern

	This idea that there is generality in the specific is of far-reaching importance. *— Douglas Hofstadter, Gödel, Escher, Bach*

Note: Today's entry is a technical article: it isn't funny. At least not intentionally.

Update, Oct 20th 2008: I've added an Updates section, where I'll try to track significant responses, at least for a week or so. There are three entries so far.

Contents

    - Eclipse
    - JavaScript
        - Pushing it even further
        - The pattern takes shape...
    - Wyvern
    - Lisp
    - XML revisited
    - Bigtable
- Properties Pattern high-level overview
- Representations
    - Keys
        - Quoting
        - Missing keys
    - Data structures
- Inheritance
    - The deletion problem
    - Read/write asymmetry
    - Read-only plists
- Performance
    - Interning strings
    - Perfect hashing
    - Copy-on-read caching
    - Refactoring to fields
    - Refrigerator
    - REDACTED
    - Rolling your own
- Transient properties
    - The deletion problem (remix)
- Persistence

    - Query strategies
    - Backfills
- Type systems
- Toolkits
- Problems
- Further reading
- New Updates
- Final thoughts

Introduction

Today I thought I'd talk about a neat design pattern that doesn't seem to get much love: the *Properties Pattern*. In its fullest form it's also sometimes called the *Prototype Pattern*.

People use this pattern all over the place, and I'll give you a nice set of real-life examples in a little bit. It's a design pattern that's useful in every programming language, and as we'll see shortly, it's also pretty darn useful as a general-purpose persistence strategy.

But even though this pattern is near-universal, people don't talk about it very often. I think this is because while it's remarkably flexible and adaptable, the Properties Pattern has a reputation for not being "real" design or

"real" modeling. In fact it's often viewed as a something of a shameful cheat, particularly by overly-zealous proponents of object-oriented design (in language domains) or relational design (in database domains.) These well-meaning folks tend to write off the Properties Pattern as "just name/value pairs" – a quick hack that will just get you into trouble down the road.

I hope to offer a different and richer perspective here. With luck, this article might even help begin the process making the Properties Pattern somewhat fashionable again. Time will tell.

Three Great Schools of Software Modeling

Before I tell you anything else about the Properties Pattern, let's review some of the most popular techniques we programmers have for modeling problems.

I should point out that none of these techniques is tied to "static typing" or "dynamic typing" per se. Each of these modeling techniques can be used with or without static checking. The modeling problem is *orthogonal* to static typing, so regardless of your feelings about static checking, you should recognize the intrinsic value in each of these techniques.

Class Modeling

You know all about this one. Class-based OO design is the 800-pound gorilla of domain modeling these days. Its appeal is that it's a natural match for the way we already model things in everyday life. It can take a little practice at first, but for most people class modeling quickly becomes second nature.

Although the *industry* loves OO design, it's not especially well liked as an academic topic. This is because OO design has no real mathematical foundation to support it — at least, not until someone comes along and creates a formal model for side effects. The concepts of OOP stem not from mathematics but from fuzzy intuition.

This in some sense explains its popularity, and it also explains why OOP has so many subtly different flavors in practice: whether (and how) to support multiple inheritance, static members, method overloading vs. rich signatures, and so on. Industry folks can never quite agree on what OOP is, but we love it all the same.

Relational Modeling

Relational database modeling is a bit harder and takes more practice, because its strength stems from its mathematical foundation. Relational modeling *can* be intuitive, depending on the problem domain, but most people would agree that it is not *necessarily* so: it takes some real skill to learn how to model arbitrary problems as relational schemas.

Object modeling and relational modeling produce very different designs, each with its strengths and weaknesses, and one of the trickiest problems we face in our industry has always been the object-relational mapping (ORM) problem. It's a big deal. Some people may have let you to believe that it's simple, or that it's automatically handled by frameworks such as Rails or Hibernate. Those who know better know just how hard ORM is in real-world production schemas and systems.

XML Modeling

XML provides yet another technique for modeling problems. Usually XML is used to model *data*, but it can also be used to model code. For instance, XML-based frameworks such as Apache Ant and XSLT offer computational facilities: loops or recursion, conditional expressions and setting variables.

In many domains, programmers will decide on an XML representation before they've thought much about the class model, because for those domains XML actually offers the most convenient way of thinking about the problem.

The kinds of data that work well with XML modeling tend to be poorly suited for relational modeling, and vice-versa, with the practical result that XML/relational mapping is almost as infamously thorny as O/R mapping.

And as for XML/OO mapping, most of us tend to treat it as a more or less solved problem. However, in practice there are several competing ways of doing XML/OO mapping. The W3C DOM and SAX enjoy the broadest use, but they are both sufficiently cumbersome that alternatives such as JDom and REXML (among others) have gained significant followings.

I mention this not to start a fight, but only to illustrate that XML is a third modeling technique in its own right. It has both natural resonances and surfaces of friction with both relational design and OO design, as one might expect.

Other schools

I'm not claiming that these three modeling schools are the

only schools out there – far from it! Two other obvious candidates are Functional modeling (in the sense of Functional Programming, with roots in the lambda calculus) and Prolog-style logical modeling. Both are mature problem-modeling strategies, each with its pros and cons, and each having varying degrees of overlap with other strategies. And there are still other schools, perhaps dozens of them.

The important takeaway is that none of these modeling schools is "better" than its peers. **Each one can model essentially any problem.**

There are tradeoffs involved with each school, by definition — otherwise all but one would have disappeared by now.

Finding the sweet spot

Sometimes it makes sense to use multiple modeling techniques in the same problem space. You might do a mixed XML/relational data design, or a class-based OO design with Functional aspects, or embed a rules engine in a larger system.

Choosing the right technique comes down to **convenience**. For any given real-world problem, one or

two modeling schools are likely to be the most convenient approaches. Exactly which one or two depends entirely on the particulars of the problem.

By *convenient*, I mean something different from what you might be thinking. To me, a convenient design is one that is convenient for the *users* of the design. And it should also be convenient to *express*, in the sense of minimalism: all else being equal, a smaller design beats a big one. One way of looking at this is that the design should be convenient for itself!

Unfortunately, most programmers (myself included) tend to use exactly the wrong definition of convenience: they choose a modeling technique that is convenient for themselves. If they only have experience in one or two schools, guess which techniques they'll jump to for *every* problem they face?

This problem rears its head throughout computing. There's always a "best" tool for any job, but if programmers don't know how to use it, they'll choose an inferior tool because they think their schedule doesn't permit a learning curve. In the long run they're hurting their schedules, but it's hard to see that when you're down in the trenches.

Modeling schools are just like programming languages, web frameworks, editing environments and many other

tools: you won't know how to pick the right one unless you have a reasonably good understanding of all of them, and preferably some practice with each.

The important thing to remember is that *all* modeling schools are "first class" in the sense of being able to represent any problem, and *no* modeling school is ideal for every situation. Just because you are most comfortable solving a problem using a particular strategy does not mean that it is the ideal solution to the problem. The best programmers aim to master all available techniques, giving them a better chance at making the right choices.

Property Modeling

With this context in mind, I claim that the Properties Pattern is yet another kind of domain modeling, with its own unique strengths and tradeoffs, distinct from all the other modeling schools I've mentioned. It is their first-class peer, inasmuch as it is capable of modeling the same broad set of problem domains.

After we've finished talking about the Properties Pattern in exhausting detail, I think I'll have convinced you of the pattern's status as a major school of modeling. Hopefully you'll also start to have a feel for the kinds of problems it's

well-suited to solve – sometimes more so than other schools, even your current favorite.

But before we dive into technical details, let's take a brief peek at a fascinating comparison of Property-based modeling to class-based OO design. It's a non-technical argument that I think has some real force behind it.

Brains and Thoughts

Douglas Hofstadter has spent a lifetime thinking about *the way we think*. He's written about it perhaps more than anyone else in the past century. Even if someone out there has beaten him in sheer quantity of words on the subject, nobody has come close to rivaling his style or his impact on programmers everywhere.

All of his books are wonderfully imaginative and are loads of fun to read, but if you're a programmer and you haven't yet read Gödel, Escher, Bach: An Eternal Golden Braid (usually known as "GEB"), then I envy you: you're in for a real treat. Get yourself a copy and settle in for one of the most interesting, maddening, awe-inspiring and just plain *fun* books ever written. The Pulitzer Prize it won doesn't nearly do it justice. It's one of the greatest and most unique works of imagination of all time.

Hofstadter made a compelling argument in GEB (thirty years ago!) that property-based modeling is *fundamental to the way our brains work*. In Chapter XI ("Brains and Thoughts"), there are three little sections titled Classes and Instances, The Prototype Principle, and The Splitting-off of Instances from Classes that together form the conceptual underpinnings of the Properties Pattern. In these little discussions Hofstadter explains how the Prototype Principle relates to classic class-based modeling.

I wish I could reproduce his discussion in full here — it's only three pages — but I'll have to just encourage you to go read it instead. His thesis is this:

> ***The most specific event can serve as a general example of a class of events.***

Hofstadter offers several supporting examples for this thesis, but I'll paraphrase one of my all-time favorites. It goes more or less as follows.

Imagine you're listening to announcers commenting on an NFL (American football) game. They're talking about a new rookie player that you don't know anything about. At this point, the rookie – let's say his name is L.T. – is just an instance of the class "football player" with no differentiation.

The announcers mention that L.T. is a running back: a bit like Emmitt Smith in that he has great speed and balance, and he's great at finding holes in the defense.

At this point, L.T. is basically an "instance" of (or a clone of) Emmitt Smith: he just inherited all of Emmitt's properties, at least the ones that you're familiar with.

Then the announcers add that L.T. is also great at catching the ball, so he's sometimes used as a wide receiver. Oh, and he wears a visor. And he runs like Walter Payton. And so on.

As the announcers add distinguishing attributes, L.T. the Rookie gradually takes shape as a particular entity that relies less and less on the parent class of "football player". He's become a very, very specific football player.

But here's the rub: even though he's a specific instance, you can now use him as a class! If Joe the Rookie comes along next season, the announcers might say: "Joe's a lot like L.T.", and just like that, Joe has inherited all of L.T.'s properties, each of which can be overridden to turn Joe into his own specific, unique instance of a football player.

This is called *prototype-based modeling*: Emmitt Smith was a prototype for L.T., and L.T. became a prototype for Joe, who in turn can serve as the prototype for someone else. Hofstadter says of The Prototype Principle:

> **"This idea that there is generality in the specific is of far-reaching importance."**

Again, it's a three-page discussion that I've just skimmed here. You should go read it for yourself. Heck, you should read the whole book: it's one of the greatest books ever written, period, and every programmer ought to be familiar with it.

Hopefully my little recap of Hofstadter's argument has convinced you that my calling it the "Universal Design Pattern" might just *possibly* be more than a marketing trick to get you to read my blog, and that the rest of this article is worth a look.

The Properties Pattern is unfortunately big enough to deserve a whole book. Calling an entire school of modeling a "design pattern" is actually selling it short by a large margin.

Hence, if this article seems excruciatingly long, it's because I've tried to cram a whole book into a blog, as I often do. But look on the bright side! I've saved you a bunch of time this way. This will go much faster than reading a whole book.

Even so, don't feel bad if it takes a few sittings to get through it all. It's still a lot of information. I considered splitting it into 3 articles, but instead I just cut about half of the material out. (Jeff and Joel: seriously. I cut 50%.)

Who uses the Properties Pattern?

I assume I've already convinced you that this pattern is worth learning about, or you'd have left by now. I'm showing you these use cases not to draw you in, but to show you some of the very different ways the pattern can be used.

We could probably find hundreds of examples, but I'll focus on just a handful of real-world uses that I hope will illustrate just how widespread and flexible it is.

Before we start, there are two things to keep in mind. The first is that people have different names for this pattern, because even though it's quite commonplace, there hasn't been much literature on it. One paper calls it Do-It-Yourself Reflection. Another article calls it Adaptive Object Modeling. See Further reading for the few links I could dig up.

Whatever name you use, once you know how to look for it, you'll start seeing this pattern everywhere, wearing

various disguises. Now that we have a common name for it, it should be easier to spot in the wild.

The second thing to keep in mind is that the Properties Pattern *scales up*: you can choose how much to use it in your system. It can be anything from simple property lists attached to a few of your classes to make them user-annotatable, up through a full-fledged prototype-based framework that serves as the foundation for modeling everything in your system.

So our examples will range from small ones to very big ones.

Eclipse

One nice small-scale example of the pattern is the Eclipse Java Development Tools (JDT): a set of classes that model the Java programming language itself, including the abstract syntax tree, the symbol graph and other metadata. This is used by the Eclipse backend to do all the neat magic it does with your Java code, by treating Java source code as a set of data structures.

You can view the javadoc for this class hierarchy at help.eclipse.org. Click on JDT Plug-in Developer Guide, then Programmer's Guide, then JDT Core, then

`org.eclipse.jdt.core.dom`. This package defines strongly-typed classes and interfaces that Eclipse uses for modeling the Java programming language itself.

If you click through any class inheriting from `ASTNode` you'll see that it has a property list. `ASTNode`'s javadoc comment says:

> "Each AST node is capable of carrying an open-ended collection of client-defined properties. Newly created nodes have none. `getProperty` and `setProperty` are used to access these properties."

I like this example for several reasons. First, it's a very simple use of the Properties pattern. It doesn't muck around with prototypes, serialization, metaprogramming or many of the other things I'll talk about in a little bit. So it's a good introduction.

Second, it's placed smack in the middle of a very, very strongly-typed system, showing that the Properties pattern and conventional statically-typed classed-based modeling are by no means mutually exclusive, and can complement one another nicely.

And third, their property system *itself* is fairly strongly typed: they define a set of support classes such as `StructuralPropertyDescriptor`,

`SimplePropertyDescriptor`, and `ChildListPropertyDescriptor` to help place some constraints on client property values. I'm not a huge fan of this approach myself, since I feel it makes their API fairly heavyweight. But it's a perfectly valid stylistic choice, and it's useful for you to know that you can implement the pattern this way if you so choose.

JavaScript

At the other end of the "how far to go with it" spectrum we have the JavaScript programming language, which places the Prototype Principle and Properties Pattern at the very core of the language.

People love to lump dynamic languages together, and they'll often write off JavaScript as some sort of inferior version of Perl, Python or Ruby. I was guilty of this myself for over a decade.

But JavaScript is substantively different from most other dynamic languages (even Lisp), because it has made the Properties Pattern its central modeling mechanism. It borrowed this heritage largely from a language called Self, and some other modern languages (notably Io and one other language that I'll talk about below) have also chosen prototypes and properties over traditional classes.

In JavaScript, every user-interactible object in the system inherits from `Object`, which has a built-in property list. Prototype inheritance (think back to our example of the Emmitt Smith instance having been the prototype for the L.T. instance) is a first-class language mechanism, and JavaScript offers several kinds of syntactic support for accessing properties and declaring property lists (as "object literals").

JavaScript is often accurately described as the world's most misunderstood programming language. Armed with our newfound knowledge, we can start to see JavaScript in a new light. To use JavaScript effectively, you need to gain experience with a whole new School of Modeling. If you simply try to use JavaScript as a substitute for (say) Java or Python, you'll encounter tremendous friction.

Since most of us have precious little actual experience with property-based modeling, this is exactly what happens, and it's no wonder JavaScript gets a bad rap.

Pushing it even further

In addition to how centrally you want to use the Properties pattern in your system, you can also decide how *recursive* to make it: do your properties have explicit meta-properties? Do you have metaprogramming hooks? How

much built-in reflection do you offer?

JavaScript offers a few metaprogramming hooks. One such hook is the recently-introduced `__noSuchMethod__`, which lets you intercept a failed attempt to invoke a nonexistent function-valued property on an object.

Unfortunately JavaScript does not offer as many hooks as I'd like. For instance, there is no corresponding *__noSuchField__* hook, which limits the overall flexibility somewhat. And there are no standard mechanisms for property-change event notification, nor any reasonable way to provide such a mechanism. So JavaScript gets it mostly right, but it stops short, possibly for performance reasons, of offering a fully-extensible metaprogramming system such as those offered by SmallTalk and to some extent, Ruby.

The pattern takes shape...

Before we move on to other uses of the Property Pattern, let's put JavaScript (and its central use of the pattern) into perspective, by comparing it to another successful language.

First: JavaScript is not my favorite language. I've done a *lot* of JavaScript programming over the past 2 years or so, both client-side and server-side, so I'm as familiar with it

as I am with any other language I've used.

JavaScript in its current incarnation is not the best tool for many tasks. For instance, it's not great for building APIs, and it's not great for Unix scripting the way Perl and Ruby are. It has no library or package system, no namespaces, and is missing many other modern conveniences. If you're looking for a general-purpose language, JavaScript leaves you wanting.

But JavaScript *is* the best tool for many other tasks. As just one example, JavaScript is an *outstanding* language for writing unit tests — both for itself, and also for testing code in other languages. Being able to use the Properties Pattern to treat every object (and class) as a bag of properties makes the creation of mock objects a dream come true. The syntactic support for object literals makes it even better. You don't need any of the silly frameworks you see coming from Java, C++ or even Python.

And JavaScript is one of the two best *scripting languages* on the planet, in the most correct sense of the term "scripting language": namely, languages that were designed specifically to be embedded in larger host systems and then used to manipulate or "script" objects in the host system. This is what JavaScript was designed to do. It's reasonably small with some optional extensions, it has a reasonably tight informal specification, and it has a carefully crafted interface for surfacing host-system

objects transparently in JavaScript.

In contrast, Perl, Python and Ruby are huge sprawls, all trying (like C++ and Java) to be the best language for every task. The only other mainstream language out there that competes with JavaScript for scripting arbitrary host systems is Lua, famous for being the scripting language of choice for the game industry.

And wouldn't you know it, Lua is *also* a language that uses the Properties Pattern as its central design. Its central `Table` structure is remarkably similar to JavaScript's built-in `Object`, and Lua also uses prototypes rather than classes.

So the world's two most successful scripting languages are prototype-based systems. Is this just a cosmic coincidence? Or is it possible that a suitably designed class-based language could have been just as successful?

It's hard to say. I've used Jython as an embedded scripting language for a long time, and it's worked pretty well. But I've personally come to believe that the Properties Pattern is actually better suited for *extensibility* than class-based modeling, and that prototype-based languages make better extension languages than class-based languages. That's effectively what's happening with embedded scripting: the end-users are *growing* and

extending the host system.

In fact I was convinced of it before I even knew JavaScript. Let's take a look at another interesting "Who uses it?" example: Wyvern.

Wyvern

My multiplayer game Wyvern takes the Properties Pattern quite far as well, although in some different directions than what we've discussed so far. I designed Wyvern long before I'd heard of Self or Lua, and before I'd learned anything about JavaScript. In retrospect it's amazing how similar my design was to theirs.

Wyvern is implemented in Java, but the root `GameObject` class has a property list, much like JavaScript's `Object` base class. Wyvern has prototype inheritance, but since I'd never heard of prototypes before, I called them *archetypes*. In Wyvern, any game object can be the archetype for any other game object, and property lookup and inheritance work more or less identically to the way they work in JavaScript.

I arrived at this design after scratching my head for *months* (in late 1996) over how to build the ultimate extensible game. I wanted *all* the game content to be

created by players, and I came up with dozens upon dozens of detailed use cases, in all of which I wanted players to be able to extend the game functionality in surprising new ways. In the end I arrived at a set of interleaved design patterns, including a rich command system, a rich hooks/advice system, and several other subsystems I'd love to document someday.

But the core data model was the Properties Pattern.

In some ways, Wyvern's implementation is more full-featured than JavaScript's. Wyvern offers more metaprogramming facilities, such as vetoable property change notifications, which gives in-game objects tremendous flexibility in responding to their environment. Wyvern also supports both transient and persistent properties, a scheme I'll discuss below.

On other ways, Wyvern just made different decisions. One big one is that Wyvern's property values are statically typed. The property *names* are always strings, just like in JavaScript, but the values can be various leaf types (ints, longs, booleans, strings, etc.), or functions (a trick that wasn't easy in Java), or even archetypes.

But despite the differences, Wyvern's core property-list infrastructure is a lot like that of JavaScript, Self and Lua. And it's been a design I've been fundamentally happy with for over ten years. It's met or exceeded all my original

expectations for enabling end-user extensibility, particularly in its ability to let people extend the in-game behavior on the fly, without needing to reboot. This has proven extraordinarily powerful (and popular with the players.)

Where Wyvern clearly got the pattern wrong was in its lack of support for *syntax*. As soon as I decided to use the Properties pattern centrally in my game, I should have decided to use a programming language better suited for implementing the pattern: ideally, one that supports it from the ground up.

I eventually wound up using Python (actually, Jython) for a ton of my code, and it was far more succinct and flexible than anything I wrote in Java. But I was foolishly worried about performance, and as a result I wound up writing at least half the high-level game logic in Java and piling on hundreds of thousands of lines of `getProperty` and `setProperty` code. And now the system is hard to optimize; it would have been much easier if I'd had a cleaner separation of game-engine infrastructure from "scripty" game code.

Even if I'd done the whole game in Python, I'd still have had to implement a prototype inheritance framework to enable any object to be able to serve as the prototype for any other object.

I realize I haven't really explained *why* prototype inheritance works so well, except for my brief mention of mock objects for unit testing. But to keep this article tractable, I had to delete several pages of detailed examples, such as "Chieftain Monsters" that could be programmatically constructed by adding a few new properties to any existing monster

When I told you this pattern was big enough for a book, I meant a *big* book. Without the examples handy, all I can do is say that using JavaScript/Rhino (or Lua, once it became available on the JVM) might have made my life easier. Or heck, writing my own language might have been the best choice for a system that large and ambitious.

In any case, live and learn. It's a lot of code, but Wyvern is still a properties-based, prototype-based system, and it has amazing open-ended flexibility as a result.

We've been through the two big examples now (Wyvern and JavaScript). I'll close this "Who Uses It" section with just a few more key examples.

Lisp

Lisp features a small-scale example of the Properties Pattern: it has property lists for symbols. Symbols are

first-class entities in Lisp. They're effectively the names in your current namespace, like Java's fully-qualified class names.

If Java classes all had property lists, it would still be a small-scale instantiation of the Properties pattern, but it would open up an awful lot of new design possibilities to Java programmers. Similarly, Lisp stops short of making *everything* have a property list, but to the extent it offers property lists they're exceptionally useful design tools.

Emacs Lisp actually makes heavy use of the Properties Pattern, inasmuch as essentially every one of its thousands of configuration settings is a property in the global namespace. It supports the notion of transient vs. persistent properties, and it offers a limited form of property inheritance via its buffer-local variables.

Unfortunately Emacs doesn't support any notion of prototypes, and in fact it doesn't have any object-orientation facilities at all. Sometimes I want to model things in Emacs using objects with flexible property lists, and at such times I find myself wishing I were using JavaScript. But even without prototypes, Emacs gains significant extensibility from its use of properties for data representation.

Keep in mind that there are, of course, big tradeoffs to make when you're deciding how much to use the

Properties pattern; I'll discuss them in a bit. I'm not criticizing *any* of the systems or languages here for the choices they've made; all of them have been improved by their use of this pattern, regardless of how far they decided to take it.

XML revisited

Earlier I described XML as a first-class modeling school. Now that we have more context, it's possible to view XML as being an instantiation of the Properties Pattern, inasmuch as it uses the pattern as part of its fundamental structure.

XML's view of the pattern is two-dimensional: properties can take the form of either attributes or elements, each kind having different syntactic and semantic restrictions. Many folks have criticized XML for the unnecessary complexity of this redundant pair of property subsystems, but in the end it doesn't really matter much, since two ways to model properties is still better than zero ways.

So far we've seen various policies around static checking: Eclipse (strong/mandatory), JavaScript/Lua (very little), and Wyvern (moderate).

XML offers what I think is the ideal policy, which is that it

lets you decide for yourself. During your initial domain modeling (the "prototyping" phase — a term now loaded with Even More Delicious Meaning), you can go with very weak typing, opting for nothing more than simple well-formedness checks. And for many problems, this is as much static checking as you'll ever need, so it's nice that you have this option.

As some of your models become more complex, you can choose to use a DTD for extra validation. And if you need a really heavy-duty constraint system, you can migrate up to a full-fledged XML Schema or Relax NG schema, depending on your needs.

XML has proven to be a very popular modeling tool for Java programmers in particular — more so than for the dynamic language communities. This is because Java offers essentially zero support for the Properties Pattern. When Java programmers need access to the pattern, the easiest approach is currently to use XML.

The Java/XML combination has proven reasonably powerful, despite the lack of syntactic integration and numerous other impedance mismatches. Using XML is still often preferable to modeling things with Java classes. Even Eclipse's AST property lists might have been better modeled using XML and a DOM: it would have been less work, and the interface would have been more familiar. And as for Apache Ant: JSON-style JavaScript objects for

build files would have been exactly what they needed, but by the time they'd realized they needed a plug-in system, the damage was done.

As Mozilla Rhino becomes better documented, and as more Java programmers begin to appreciate the usefulness of JSON as a lightweight alternative to XML, JavaScript may begin to close the gap. Rhino provides Java with the Properties Pattern much more seamlessly than any XML solution. I've already mentioned that it's superior (even to XML) for unit tests and representing mock test data.

But it goes deeper than unit testing. Every sufficiently large Java program, anything beyond medium-sized, needs a scripting engine, whether the authors realize it or not. Programs often have to grow to the size of browsers or spreadsheets or word processors before the authors finally realize they need to offer scripting facilities, but in practice, even small programs can immediately benefit from scripting. And XML doesn't fit the bill. It's yet another example of programmers choosing a School of Modeling because they know it, rather than learning how to use the right tool for the job.

So it goes.

Bigtable

Last example: Google's Bigtable, which provides a massively scalable, high performance data store for many Google applications (some of which are described in the paper – click the link if you're curious.) This particular instantiation of the Properties Pattern is a multidimensional table structure, where the keys are simple strings, and the leaf values are opaque blobs.

Hardcore relational data modelers will sometimes claim that large systems will completely degenerate in the absence of strong schema constraints, and that such systems will also fail to perform adequately. Bigtable provides a nice counterexample to these claims.

That said, explicit schemas *are* useful for many domains, and I'll talk more about how they relate to the Properties Pattern in a bit.

This would probably be a good time to mention Amazon's Simple Storage Service, but I don't know anything about it. I've heard they use name-value pairs.

In any case, I hope these examples (Eclipse AST classes, JavaScript, Wyvern game objects, Lisp symbols, XML and HTML, and Bigtable) have convinced you that the Properties pattern is ubiquitous, powerful, and multifaceted, and that it should be part of any programmer or designer's lineup.

Let's look in more depth at how it's implemented, its trade-offs, and other aspects of this flexible design strategy.

Properties Pattern high-level overview

At a high level, every implementation of the Properties Pattern has the same core API. It's the core API for *any* collection that maps names to values:

- **get(name)**
- **put(name, value)**
- **has(name)**
- **remove(name)**

There are typically also ways to iterate over the properties, optionally with a filter of some sort.

So the simplest implementation of the Properties Pattern is a Map of some sort. The objects in your system are Maps, and their elements are Properties.

The next step in expressive power is to reserve a special property name to represent the (optional) parent link. You can call it "parent", or "class", or "prototype", or "mommy", or anything you like. If present, it points to another Map.

Now that you have a parent link, you can enhance the semantics of **get**, **put**, **has** and **remove** to follow the parent pointer if the specified property isn't in the object's list. This is largely straightforward, with a few catches that we'll discuss below. But you should be able to envision how you'd do it without too much thought.

At this point you have a full-fledged Prototype Pattern implementation. All it took was a parent link!

From here the pattern can expand in many directions, and we'll cover a few of the interesting ones in the remainder of this article.

Representations

There are two main low-level implementation considerations: how to represent property keys, and what data structure to use for storing the key/value pairs.

Keys

The Properties pattern almost always uses String keys. It's possible to use arbitrary objects, but the pattern becomes more useful with string keys because it trivially enables

prototype inheritance. (It's tricky to "inherit" a property whose key is some opaque blob - we usually think of inheritance as including a set of named fields from the parent.)

JavaScript permits you to use arbitrary objects as keys, but what's really happening under the covers is that they're being cast to strings, and they lose their unique identity. This means JavaScript `Object` property lists cannot be used as be a general-purpose hashtable with arbitrary unique objects for keys.

Some systems permit both strings and numbers as keys. If your keys are positive integers, then your Map starts looking an awful lot like an Array. If you think about it, Arrays and Maps share the same underlying formalism (a surjection, not to put too fine a point on it), and in some languages, notably PHP, there isn't a user-visible difference between them.

JavaScript permits numeric keys, and allows you to specify them as either strings or numbers. If your object is of type Array, you can access the numeric keys via array-indexing syntax.

Quoting

JavaScript syntax is especially nice (compared to Ruby

and Python) because it allows you to use *unquoted* keys. For instance, you can say

```
var person = {
  name: "Bob",
  age: 20,
  favorite_days: ['thursday', 'sunday']
}
```

and the symbols *name*, *age* and *favorite_days* are NOT treated as identifiers and resolved via the symbol table. They're treated exactly as if you'd written:

```
var person = {
  "name": "Bob",
  "age": 20,
  "favorite_days": ['thursday', 'sunday']
}
```

You also have to decide whether to require quoting *values*. It can go either way. For instance, XML requires attribute values to be quoted, but HTML does not (assuming the value has no whitespace in it).

Missing keys

You will need to decide how to represent "property not present". In the simplest case, if the key isn't in the list, the property is not there (but see Inheritance further on).

If a property is frequently removed and re-added, it may

make sense to leave the key in the list with a null value. In some systems, you may need `null` to be a valid property value, in which case you'd need to use some other distinguished (and reserved) value for this micro-optimization to work.

Data structures

The simplest property-list implementation is a linked list. You can either have the alternating elements be the keys and values (Lisp does this), or you can have each element be a struct containing pointers to the key and value.

The linked list implementation is appropriate when:

- you're just using the pattern to allow user annotations on object instances
- you don't expect many such annotations on any given instance
- you're not incorporating inheritance, serialization or meta-properties into your use of the pattern

Logically a property list is an unordered set, not a sequential list, but when the set size is small enough a linked list can yield the best performance. The performance of a linked list is O(N), so for long property lists the performance can deteriorate rapidly.

The next most common implementation choice is a hashtable, which yields amortized constant-time find/insert/remove for a given list, albeit at the cost of more memory overhead and a higher fixed per-access cost (the cost of the hash function.)

In most systems, a hashtable imposes too much overhead when objects are expected to have only a handful of properties, up to perhaps two or three dozen. A common solution is to use a hybrid model, in which the property list begins life as a simple array or linked list, and when it crosses some predefined threshold (perhaps 40 to 50 items), the properties are moved into a hashtable.

Note that we'll often refer to property sets as "property lists" (or "plists" for short), because they're so often implemented as lists. But it's fairly unusual for the order to matter. In the rare cases when it matters, there are usually two possibilities: the names need to be kept in insertion order, or they need to be sorted.

If you need constant-time access and want to maintain the insertion order, you can't do better than a LinkedHashMap, a truly wonderful data structure. The only way it could possibly be more wonderful is if there were a concurrent version. But alas.

If you need to impose a sort order on property names, you'll want to use an ordered-map implementation,

typically an ordered binary tree such as a splay tree or red/black tree. A splay tree can be a good choice because of the low fixed overhead for insertion, lookup and deletion, but with the tradeoff that its theoretical worst-case performance is that of a linked list. A splay tree can be especially useful when properties are not always accessed uniformly: if a small subset M of an object's N properties are accessed most often, the amortized performance becomes O(log M), making it a bit like an LRU cache.

Note that you can get a poor-man's splay tree (at least, the LRU trick of bubbling recent entries to the front of the list) using a linked list by simply moving any queried element to the front of the list, a constant-time operation. It's surprising that more implementations don't take this simple step: an essentially free speedup over the lifetime of most property lists.

Inheritance

With prototype inheritance, each property list can have a parent list. When you look for a property on an object, first you check the object's "local" property list, and then you look in its parent list, on up the chain.

As I described in the Overview, the simplest approach for

implementing inheritance is to set aside a name for the property pointing to the parent property list: "prototype", "parent", "class" and "archetype" are all common choices.

It's unusual (but possible) to have a multiple-inheritance strategy in the Properties pattern. In this case the parent link is a list rather than a single value, and it's up to you to decide the rules for traversing the parent chains during lookups.

The algorithm for inherited property lookup is simple: *look in my list, and if the property isn't there, look in my parent's list. If I have no parent, return* `null`. This can be accomplished recursively with less code, and but it's usually wiser to do it iteratively, unless your language supports tail-recursion elimination. Property lookups can be the most expensive bottleneck in a Properties Pattern system, so thinking about their performance is (for once) almost never premature.

The deletion problem

If you delete a property from an object, you usually want subsequent checks for the property to return "not found". In non-inheritance versions of the pattern, to delete a property you simply remove its key and value from the data structure.

In the presence of inheritance the problem gets trickier, because a missing key does *not* mean "not found" – it means "look in my parent to see if I've inherited this property."

To make the problem clearer, assume you have a prototype list called Cat with a property named "friendly-to-dogs", whose value defaults to the boolean `true`. Let's say you have a specific cat instance named Morris, whose prototype is Cat:

```
var Cat = {
  friendly_to_dogs: true
}

var Morris = {
  prototype: Cat
}
```

Let's say Morris has a nasty run-in with a dog, and now he hates all dogs, so we want to make a runtime update to his friendly-to-dogs property. Our first idea might be to delete the property, since a missing key or null value are often interpreted as `false` in a boolean context. (This is true even in class-based languages like C++ or Java, in which a `hasFooBar` function will return `true` if the internal `fooBar` field is non-`null`.)

However, Morris does not have a copy of "friendly-to-dogs" in his local list: he inherits it from Cat. So if your

`deleteProperty` method does nothing but delete the property from the local list, he will continue to inherit "friendly-to-dogs", which will irk him (and you) endlessly until you figure out where the bug is.

You can't delete "friendly-to-dogs" from the Cat property list, or all of your cats will suddenly become dog-haters, and you'll have outright war on your hands. (Note that in some settings this is *exactly* what you want to do, illustrating the inherent universal trade-off between flexibility and safety.)

The solution for Morris is to have a special "NOT_PRESENT" property value that `deleteProperty` sets when you delete a property that would otherwise be inherited. This object should be a flyweight value so that you can check it with a pointer comparison.

So to account for deletion of inherited properties, we have to modify our property-lookup algorithm to look in the local list for (a) a missing key, (b) a null value, or (c) the NOT_PRESENT tag. If any of these apply, the property is considered not present on the object. [Note: the semantics of null values are up to the system designer. You don't have to make `null` values mean "not there." Either way is fine.]

Read/write asymmetry

One logical consequence of prototype inheritance as we've defined it is that reads and writes work differently. In particular, if you read an inherited property, it gets the value from an ancestor in the prototype chain. But if you *write* an inherited property, it sets the value in the object's local list, not in the ancestor.

To illustrate, let's add a "night-vision" property to our Cat prototype. Its value is expected to be an integer representing how well the cat can see in the dark. Let's say that the default value is 5, but our hero Morris has been eating his carrots, so we want to set his "night-vision" property value to 7.

The `setProperty` code does not need to check the parent chain: it simply adds the key/value pair {"night-vision", 7} to Morris's local property list. If we set the property on Cat, then all cats would have Morris's super-vision, which isn't what we want.

This asymmetry is normal. Back in our L.T. / Emmitt Smith example, when we were adding properties to L.T., we didn't want to modify Emmitt! It's just how the pattern works: you override inherited values by adding local properties, even when the override is a deletion.

Read-only plists

Many implementations of the pattern offer "frozen" property lists. Sometimes (e.g. for debugging) it's useful to flag an entire property list as read-only. Ruby supports this via the "freeze" method on the built-in root `Object` class. In any sufficiently large, robust implementation of the Properties pattern, you should include the option to freeze your property lists.

If you offer a "freeze" function, you should think about whether you want to offer a "thaw" as well. The decision depends on whether you want to offer programmers additional protection, or you just want to lock them up and throw away the key.

My personal view is that Java programmers tend to overuse the "freeze" function when they start with Ruby. For that matter, they tend to overuse "final" in Java. I mentioned before the trade-off between *flexibility* and *safety*. When you use the Properties Pattern, you're consciously choosing flexibility over safety, and in many domains this is the right choice. In fact, safety can be viewed as a kind of optimization: something that should ideally be layered on, functioning behind the scenes rather than being interleaved with the user APIs and flexible data model.

A nice (and simple to implement) compromise on safety and flexibility is to offer a `ReadOnly` property attribute, as

JavaScript does. There are certain properties (such as the parent pointer) that are less likely to need to change as the system evolves, so it's probably OK to lock them down early on. Doing this on a property-by-property basis is much less draconian. Even better, you should consider making the `ReadOnly` property attribute non-inheritable, so that subtypes can choose their own policies without compromising the integrity of the supertypes.

We're more or less done with inheritance: it's not very complicated. There are a few other inheritance-related design issues that I'll cover in upcoming sections.

Performance

Performance is one of the biggest trade-offs of using the Properties Pattern. Many engineers are so concerned with performance (and its attendant paradoxes and fallacies) that they refuse to consider using the Properties pattern, regardless of the situation.

As it happens, the pattern's performance can be improved and mitigated in several clever ways. I won't cover all of them here, but I'll touch on some of the classics and one or two new approaches.

Interning strings

Make sure your string keys are interned. Most languages provide some facility for interning strings, since it's such a huge performance win. Interning means replacing strings with a canonical copy of the string: a single, immutable shared instance. Then the lookup algorithm can use pointer equality rather than string contents comparison to check keys, so the fixed overhead is much lower.

The only downside of interning is that it doesn't help much when you're constructing a property name on the fly, since you still need to hash the string to intern it.

That's not much of a downside, so as a rule, you should always intern your keys. A large percentage of property names in any system are accessed as string literals from the source code (or are read from a configuration file and can be interned all at once when the file is read), and interning works in these common cases.

Corollary: don't use case-insensitive keys. It's performance suicide. Case-insensitive string comparison is really slow, especially in a Unicode environment.

Perfect hashing

If you know all the properties in a given plist at compile-time (or at runtime early on in the life of the process), then you might consider using a "perfect hash function generator" to create an ideal hash function just for that list. It's almost certainly more work than it's worth unless your profiler shows that the list is eating a significant percentage of your cycles. But such generators (e.g. gperf) do exist, and are tailor-made for this situation.

Perfect hashing doesn't conflict with the extensible-system nature of the Properties pattern. You may have a particular set of prototype objects (such as your built-in monsters, weapons, armor and so on) that are well-defined and that do not typically change during the course of a system session. Using a perfect hash function generator on them can speed up lookups, and then if any of them is modified at runtime, you just fall back to your normal hashing scheme for that property list.

Copy-on-read caching

If you have lots of memory, and your leaf objects are inheriting from prototype objects that are unlikely to change at runtime, you might try copy-on-read caching. In its simplest form, whenever you read a property from the parent prototype chain, you copy its value down to the

object's local list.

The main downside to this approach is that if the prototype object from which you copied the property ever changes, your leaf objects will have the now-incorrect old value for the property.

Let's call copy-on-read caching "plundering" for this discussion, for brevity. If Morris caches his prototype Cat's copy of the "favorite-food" property (value: "9Lives"), then Morris is the "plunderer" and Cat is the plundered object.

The most common workaround to the stale-cache problem is to keep a separate data structure mapping plundered objects to their plunderers. It should use weak references so as not to impede garbage collection. (If you're writing this in C++, then may God have mercy on your soul.) Whenever a plundered object changes, you need to go through the plunderers and remove their cached copy of the property, assuming it hasn't since then changed from the original inherited value.

That's a lot of stuff to keep track of, so plundering is a strategy best used only in the direst of desperation. But if performance is your key issue, and nothing else works, then plundering may help.

Refactoring to fields

Another performance speedup technique is to take your N most commonly used properties and turn them into instance variables. Note that this technique is only available in languages that differentiate between instance variables and properties, so it would work in Java but not in JavaScript.

This optimization may sound amazingly attractive, especially during your first round of performance optimizations. Be warned: this approach is fraught with pitfalls, so (as with nearly all performance optimizations) you should only use it if your profiler proves that the benefits will outweigh the costs.

The first cost is API incompatibility: suddenly instead of accessing all properties uniformly through a single `getProperty/setProperty` interface, you now have specific fields that have their own getters and setters, which could potentially have massive impact in your system (since, after all, these are the most commonly-accessed properties). And unless you're using Emacs, your refactoring editor probably isn't smart enough to do rename-method constrained on argument value.

You can mitigate the API problem by continuing to go through the `get/setProperty` interface, and have them

check the argument to see if it's one of the hardwired fields. This will result in an increasingly large switch-statement (or equivalent), so you're trading API code maintenance for API simplicity. It also slows down the field access considerably, which offsets the performance gain from using a field.

The next cost is system complexity: you have twice as many code paths through *every* part of your system that deals with the Properties pattern. Does inheritance still work the same? What about serialization? Transient properties? Metaprogramming? What about your user interface for accessing and modifying property lists? You face a huge code-bloat problem when you split property access into two classes: plist properties and instance variables.

The next cost is accidentally getting it wrong: how do you know what the most-accessed properties are? You may do some runtime profiling and see that it's one set, but over time the characteristics of your system might change in such a way that it's an entirely different set. Realistically you will have to instrument your system to keep track, on a regular basis, of which properties are accessed at a rate beyond your tolerance threshold, so you can convert them to fields as well. This isn't a one-off optimization.

But all these costs pale in comparison to the big one, which is extensibility: instance variables are set in stone. It

will be a vast amount of work to try to give them parity with your plist properties: change-list notifications, the ability to override them or remove them, and so on. It's likely that you will wind up sacrificing at least some flexibility for these fields.

So use this optimization with extreme caution.

Refrigerator

The last performance optimization I'll mention is more about conserving memory than CPU. If you're worried about the overhead of a per-object property-list field, even if it's usually `null`, then you can implement property lists using a separate, external data structure.

I don't know what it's normally called, so I'm calling it the Refrigerator, since you're basically putting yellow stickies all over it. The idea is that you don't need to pay for the overhead of property lists when very few of the objects in your system will ever have one. Instead of using a field in each class with a property list, you maintain a global hashtable whose keys are object instances, and whose values are property lists.

To fetch the property list of an object, you go look to see if it's on the Refrigerator. The property list itself can follow

any of the implementation schemes I discussed in Representations above.

I first heard this idea from Damien Conway in roughly 2001 at a talk he gave. He said he was considering using it for Perl 6, and I thought it was pretty clever. I don't remember what he called it, and I don't know if he wound up using it, but consider this idea to be his gift to you. Thanks, Damien!

REDACTED

Brendan Eich came up with astoundingly clever performance optimization for the Properties Pattern, which he told me about back in January. I was ready to publish this article, but I told him I'd hold off until he blogged about his optimization. Every once in a while he'd ping me and tell me "any day now."

Brendan, it's *October*, dammit!

Rolling your own

Needless to say, I've only scratched the surface on performance optimization of the Properties pattern. You can get arbitrarily fancy. The point I'm trying to get across

is that you shouldn't despair when you discover your system is unacceptably slow after designing it to use the Properties pattern. If this happens, don't panic and throw out your flexibility – go optimize! The game of optimization can be fun and rewarding in its own right.

Just don't do it before you need it!

Transient properties

While implementing Wyvern, I discovered that making changes to a persistent property list is a wonderful recipe for creating catastrophes.

Let's say some player casts a Resist Magic spell, which boosts her "resist-magic" integer property value by, oh, 30 (thirty percent). Then, while the spell is active, the auto-saver kicks in (writing her enhanced "resist-magic" property value out to the data store along with the rest of her properties), and then the game crashes.

Voilà – the player now has permanent 30% magic resistance!

It doesn't have to be a game crash, either. Any random bug or exception condition (a database hiccup, a network glitch, cosmic rays) can induce permanence in what was

intended to be a transient change to the plist. And when you're writing a game designed to be modified at runtime by dozens of programmers simultaneously, you learn quickly to expect random bugs and exception conditions.

The solution I came up with was transient properties. Each object has (logically speaking) *two* property lists: one for persistent properties and one for transients. The only difference is that transient properties aren't written out when serializing/saving the player (or monster, or what-have-you.)

Wyvern's property-list system has typed values. I haven't talked about Properties Pattern type systems yet, but in a nutshell my property values can be ints, longs, doubles, strings, booleans, archetypes (which is basically any other game object), or "bean" (JavaBean) properties.

My early experimentation yielded the interesting rule that non-numeric transient properties *override* the persistent value, but numeric properties *combine* with (add to) the persistent value.

A simple example should suffice. If you have a persistent boolean property "hates-trolls" (and who doesn't, really?), and you accidentally ingest a Potion of Troll Love, then the potion should set a transient value of {"hates-trolls", `false`} on your character. It overrides the persistent value. There's no combining going on; it just replaces the

original.

However, for our "resist-magic" int property, if you put on a ring of 30% magic resistance, it should (by default) *add* to your current value, which may be a combination of innate resistance and resistances conferred from other magic items and spells.

This numbers-are-additive principle applied pretty uniformly across my entire code base and property corpus, so it's built into the lookup rules for Wyvern's property lists. `getIntProperty("foo")` must get both the transient and persistent (possibly inherited) values for "foo" and add them before returning the result.

I experimented with different approaches for representing transient properties. Originally I used a kind of hungarian-notation, prefixing transient property names with an @-character ("@foo") and keeping them in the same hashtable as the persistent properties. One advantage of "@" was that it was invalid character in XML attribute names, so it was originally *impossible* for me to accidentally serialize a transient property.

Eventually I migrated to keeping them in separate (lazily created) tables. This made it easier to deal with interning names (not having to prepend "@" all the time) and generally made bookkeeping easier. I don't remember all the trade-offs involved with the decision anymore (it was

about 7 years ago), so you'll have to retread that road yourself if you decide to offer transient properties in your system.

The deletion problem (remix)

Transient properties introduce their own version of deletion problem. You can't just remove the property from the transient list, since the lookup algorithm will just look for it in the persistent list. And you don't want to remove it from the persistent list; that defeats the purpose of using transients.

The solution is similar to what I did for deleting inherited properties: you insert a placeholder into the transient list saying "NOT_PRESENT", and as long as that placeholder is in the list, it's as if the object doesn't have the property.

Note that this implies the existence of two similar API calls: `removeTransientProperty` for deleting a transient property from the transient list, and `transientlyRemoveProperty` for temporarily hiding a property from the persistent list.

Persistence

Persisting property lists is a huge topic; I'll just touch on the basics.

For starters, XML and JSON (and for that matter, s-expressions) are all perfectly valid choices for serialization format. You can probably imagine how this works, so I won't beat it to death.

Text-based formats have big wins in readability, maintainability, and bootstrapping (you don't need to create special tools to read and write them).

For performance – both for network overhead and disk storage – you might want to consider designing a compressed binary format. One easy way to test whether this will be a win for you is to take the intermediate approach of gzipping your data to see how well it compresses, and whether it produces a discernable blip in performance.

Wyvern initially used a filesystem trie-like structure for storing its data, but as the number of distinct objects grew to several hundred thousand, I had to switch to a database.

You can use an RDBMS, but you're in for a world of hurt if you try to map the Properties pattern onto a relational schema. It's a tricky problem, and probably isn't one that you want to solve yourself.

I wound up using an RDBMS and just shoving the XML-serialized property list into a text/clob column, and denormalizing the twenty or thirty fields I needed for queries into their own columns. This gets me by, but isn't a happy solution.

What you really want is a hierarchical data store optimized for loose tree structures: in a word, an XML database. At the time I was designing Wyvern's persistence strategy (1998-ish), XML databases were pure vaporware, and even after a few years they were still fairly exploratory and unstable.

Today things are different, and there are many interesting options for XML databases, ranging from 100% free (e.g. Berkeley DBs) through 100% expensive (e.g. Oracle XML).

You might also look into Object databases, but I've never heard of anyone coming through that experience with anything but battle scars to show for it.

Query strategies

Querying goes hand-in-hand with persistence: once you have a bunch of objects in a data store, you'll want to ask questions about them. Producing a High Score List is a

good example: you want to compute a function of some set of properties across all the players in your database.

If you're just using the filesystem, you're stuck with grep or its moral equivalent on Windows, which is likely to be painfully slow. So don't do that.

If you're using an RDBMS, and you've serialized your property lists into a single row-per-object clob or text column, then you can use (My)SQL's LIKE and RLIKE operators, or their equivalents, to do free-text searches.

However, your property lists are likely to be hierarchical (e.g. player inventory is a bag, which has its own properties *and* collection of objects it's holding), and free-text search doesn't understand hierarchy. So this approach is really just a faster version of grep.

Querying is the biggest reason for using an XML database, since it gives you XPath and XQuery as expressive languages that work on XML data about as well (give or take) as SQL works on relational data.

Because you have the advantage of working in "these days" (2008+) as opposed to "those days" (1998), you now have the interesting option of using JavaScript/JSON and JQuery. I don't know much about it, but what little I do know seems promising.

One final approach, which may not scale very well unless you can find a way to parallelize it, is to simply load all the objects into an instance of your server, and use programmatic access to walk the objects and construct your query results manually. Although it requires some infrastructure to make it work (and to make it not crash your system, once you have enough objects), it has the major benefit of giving you a full programming language, which can be useful if you're doing a complex query and your XPath/XQuery skills aren't up to par.

Backfills

Data integrity, a.k.a. Safety, is one of the two biggest trade-offs (the other being performance) you make when you choose to use the Properties pattern. In the absence of a schema, your system is open to property-list corruption through bugs and user error (e.g. typos).

Interestingly, in big companies I've worked at that have strong schema constraints, they *still* always seem to run into data-integrity problems, so it's not clear how much the schema is really helping here. A schema can certainly help with navigability and performance, but no schema can completely avert data corruption problems.

As soon as you notice you've got bad data, you need to

do what many people in the industry term a "backfill": you have to run through all the existing data and fix the problem. Sometimes this is as simple as running a SQL update on a single column. And sometimes it involves writing a complex program that painstakingly computes the inverse of whatever bogus operation created the bad data in the first place.

And sometimes backfills require just winging it, since the lost data may be irrecoverable and you need to use heuristics to minimize the damage. No matter how you store your data and how careful you are about replicating it and backing it up, this kind of thing can happen at pretty much any scale.

The Properties pattern doesn't really introduce anything new to the backfill landscape; all the usual options apply. You'll just need to be mindful that user error (especially mis-typed property names) can make backfills a bit more commonplace, so you should plan to spend a fair amount of time developing a convenient backfill infrastructure for your system.

I should mention, embarrassing as it is, one other option, which I call "lazy backfill", and I've used it extensively. Sometimes I'll notice a data-corruption issue that needs fixing but doesn't really justify a day of my time to fix all at once. So I have a small subsystem in place for player logins and map loads: I iterate through the property lists

looking for properties that I've flagged (hardwired in the code) as "bad data", and I call helper backfill functions on the fly to fix just that property list.

This is obviously a hack, and it also imposes some minor performance overhead (probably not detectable via profiling) on logins and map loads, but I'll be honest: it's served me well, and I've fixed at least 20% of my data-corruption problems this way.

Type systems

I've already touched on this a little here and there. Eclipse's AST property lists use an interesting type system that provides a reasonable amount of metadata for each property, although (I think) it stops short of allowing properties to have their own property lists.

JavaScript properties have a small, fixed amount of metadata. Each property has a set of flags. The flags include `ReadOnly` (can't modify the value), `Permanent` (can modify the value but can't delete the key), `DontEnum` (key doesn't show up in iterators but can be read directly), and others depending on the implementation.

Wyvern has its own Java-like flavor of typed properties, largely because I implemented the system in Java long

before the advent of auto-boxing, and I needed a convenient way of dealing with primitive types vs. object types. If I were to do it all over again, I probably wouldn't go that route. I *would* want some sort of scheme for metaproperties (aka "property attributes") — perhaps in a separate per-object metaproperty-list. But I'd simplify the interface and get rid of all the primitive-typed versions of all my has/get/set inherited/persistent/transient property calls.

I won't go into any further detail about type systems, except to say that (a) you can use them to any degree you desire; there's nothing intrinsic to the Properties Pattern that precludes them, and (b) Duck Typing becomes fairly crucial to systems that are designed fully around the Properties Pattern, so if your language has any structural-typing support it'll help.

Toolkits

Wyvern has a Map Editor that allows you to create and edit objects. Since all the game objects are property lists that use the prototype inheritance pattern, the conventional JavaBean approach doesn't work, since the JavaBeans API (which is more or less designed for this problem, except with instance fields) uses Java reflection, and my properties don't have individual getters and

setters.

Wyvern wound up with something very similar to a JavaBeans property editor, except it knows how to read, write and display my GameObject property lists.

It wasn't a huge amount of work, but it's something you should keep in mind as you decide whether to use the Properties pattern in your system. If you need a GUI for object edits, you'll probably need to do some custom UI work.

Problems

I've talked about the main problems imposed by the Properties pattern: performance, data integrity, and navigability/queryability. They're all trade-offs; you're sacrificing in these areas in order to achieve big wins in flexibility and open-ended future extensibility for users you may never meet. (You also get huge wins in unit-testability and prototyping speed, but I assume these benefits are obvious enough that I don't need to dwell on them.)

One other problem is reversability: it's hard to back out of the Properties pattern once you commit to it. Once people start adding properties, especially if they're using

programmatic construction of key names, you'll have trouble on your hands if you want to try to refactor the whole system to use instance fields and getters/setters. So before you use this pattern, you should put your system through a prototyping phase (ironically enough) to determine whether it will work out as it scales.

Further Reading

I wasn't able to find much, but here are some interesting articles and papers on the subject.

Dealing with Properties [Fowler 1997]

Prototype-based programming (Wikipedia)

Do-it-yourself Reflection [Peter Sommerlad, Marcel Rüedi] (PDF)

Prototype pattern (Wikipedia)

Self programming language (Wikipedia)

Refactoring to Adaptive Object Modeling: Property Pattern [Christopher G. Lasater]

New Updates

Oct 20th 2008: The comments on the article are outstanding. People have pointed out lots of further reading, as well as well-known systems (Atom, RDF) that make extensive use of the pattern at their core. Thanks, folks! This is great stuff.

Oct 20th 2008: Martin Fowler sent me a link to a 1997 paper he wrote on the topic. It's in the Further Reading section. Well worth a read. He mentions a few important considerations that I left out:

- **The (static) dependencies problem** — your compiler can't generally help you with finding dependencies on properties, or even tell you what property names are used in your system. He suggests using a registry of permissible property names as one solution. I strongly considered that approach for Wyvern, but wound up relying on a mix of dynamic tracing and static analysis to get me the dependency graph, and it's been "accurate enough" for my needs. In particular, synthesizing property names on the fly happens about as (in)frequently as Reflection happens in Java. So for the most part the dependencies issue is as tractable as it is in Java: "tractable enough".

- **Substituting actions** Fowler suggests that it's difficult to replace an existing property access with an action. This is only true in the Java implementation (hence, true for my game). In languages like Python, Ruby and JavaScript 1.5 that support property-access syntax for getters and setters this is a non-issue.

Overall, Martin's take on the pattern is "avoid it when possible", which is sound (if conservative) advice. My take is that everyone's doing it anyway, so we should formalize it. I've used it as the central data model for my 500,000-line multiplayer game for 10 years, and I assert that the benefits vastly outweigh the problems. I also witnessed the pattern's use in Amazon's Customer Service Tools database for some 5 years, and again, the benefits vastly outweighed the downsides.

You just have to know what you're getting into before you dive in, which is sort of the point of my article.

Oct 20th 2008: Fellow Googler Joe Beda mentioned that IE4 originally supported arbitrary attributes on HTML elements, which dramatically extended the flexibility for web developers. Today, no browsers support it, though John Resig claims HTML 5 will fix this. In the meantime, developers use fake css classes and hidden elements; it's a mess. I actually deleted a pretty large rant about this

problem from the article. But yeah. It's a problem. When you don't provide the Properties Pattern to people, they find horrible workarounds, which is much worse than anything that can go wrong if you simply support it directly. (Joe mentioned that it posed serious technical problems with the cache, though, so I wouldn't assume it's trivial to add the support back in to browsers today.)

Final thoughts

I haven't covered the whole landscape for this pattern. There are concurrency issues, access-control issues (e.g. in Wyvern, some properties, such as email address, can only be read by very high-level Wizards), documentation issues, and a host of other considerations.

Let me summarize what I think are the key takeaways.

First: this is a critically important pattern. I call it the "Universal" design pattern because it is (by far) the best known solution to the problem of designing open-ended systems, which in turn translates to long-lived systems.

You might not think you're building that kind of system. But if you want your system to grow, and have lots of users, and spread like wildfire, then you *are* building exactly that kind of system. You just haven't realized it yet.

Second: even though people rarely talk much about this pattern, it's astoundingly widespread. It appears in strongly-typed systems like Eclipse, in programming and data-declarative languages, in end-user applications, in operating systems, and even in strongly typed network protocols, although I didn't talk about that use case today. (Nutshell: a team I know using CORBA got fed up and added an XML parameter to every CORBA API call, defeating the type system but permitting them to upgrade their interface without horking every existing client. Bravo!)

Third: it can perform well! Or at least, "well enough". The playing field for potential optimizations is nearly unbounded, and with enough effort you can reduce just about everything to constant time.

Finally, it's surprisingly versatile. You can use it on a very small scale to augment one teeny component of an existing system, or you can go the whole hog and use it for everything, or just about anything in between. You can start small and grow into it as you become more comfortable with the pattern.

The Properties Pattern is *not* "just" name/value pairs, although the name/value pair certainly lives at the heart of the pattern.

If you believe Hofstadter, the Properties Pattern (using the Prototype Principle) is an approach to modeling that complements class-based modeling: both are fundamental to the way our brains process symbolic information.

I suspect that if you've read this far, you'll start seeing the Properties Pattern everywhere you look. It's embedded in many other popular patterns and systems, from Dependency Injection to LDAP to DNS. As legacy systems struggle to evolve to be more user configurable and programmable, you'll see more and more of your favorite software systems (and, I hope, languages) incorporating this pattern to varying extents.

Again: I wasn't able to find much literature about this pattern. If you happen to know of books, papers or articles that expound on it in more detail, please link to them in the comments!

I hope you found this interesting and potentially useful. It was definitely more work than my typical posts, so if it doesn't go over well, I'm happy to go back to the comedy routine. We'll see.